

PCT

WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau



INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification⁶:
G11B

A2

(11) International Publication Number: **WO 99/34356**

(43) International Publication Date: **8 July 1999 (08.07.99)**

(21) International Application Number: **PCT/US98/27417**

(22) International Filing Date: **23 December 1998 (23.12.98)**

(30) Priority Data:
60/070,195 **30 December 1997 (30.12.97)** **US**

(71) Applicant: **GENESIS ONE TECHNOLOGIES, INC. [US/US];**
7470 San Clemente Place, Boca Raton, FL 33433 (US).

(72) Inventors: **O'NEIL, John, T.; Apartment 3C, 613 Ocean Drive,**
Key Biscayne, FL 33149 (US). ISRAEL, Ben; 3040 N.W.
23rd Court, Boca Raton, FL 33431 (US).

(74) Agents: **JABLON, Clark, A. et al.; Panitch Schwarze Jacobs**
& Nadel, P.C., One Commerce Square, 22nd floor, 2005
Market Place, Philadelphia, PA 19103-7086 (US).

(81) Designated States: **AL, AM, AT, AU, AZ, BA, BB, BG, BR,**
BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE,
GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ,
LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW,
MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ,
TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent
(GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent
(AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent
(AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT,
LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI,
CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

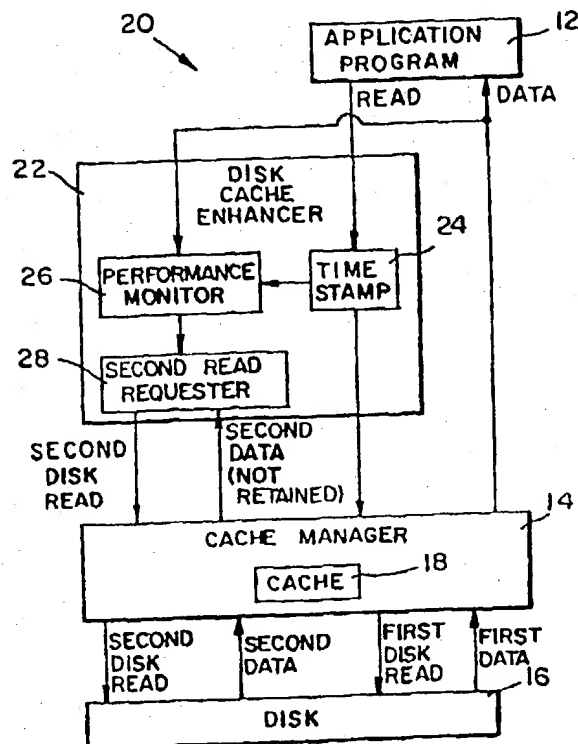
Published

*Without international search report and to be republished
upon receipt of that report.*

(54) Title: **DISK CACHE ENHANCER WITH DYNAMICALLY SIZED READ REQUEST BASED UPON CURRENT CACHE HIT RATE**

(57) Abstract

The performance of a disk cache subsystem is enhanced by dynamically sizing read requests based upon the current cache hit rate. Accordingly, the size of the read request depends upon at least one variable factor other than the size of the requested data. More specifically, the size of the read request is reduced as the cache hit rate declines, and the size of the read request is increased as the cache hit rate increases. Short-term and long-term cache hit rates are tracked. The short-term cache hit rate is used to determine the reduction in the size of the read request, and the long-term cache hit rate is used to determine the increase in the size of the read request. The read request is a read-around request formulated to obtain the immediately requested data, plus additional data which is not requested and which is located before and after the immediately requested data. When used in a Windows operating system environment, the initial read-around value is about 1 megabyte.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece	ML	Mali	TR	Turkey
BG	Bulgaria	HU	Hungary	MN	Mongolia	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MR	Mauritania	UA	Ukraine
BR	Brazil	IL	Israel	MW	Malawi	UG	Uganda
BY	Belarus	IS	Iceland	MX	Mexico	US	United States of America
CA	Canada	IT	Italy	NE	Niger	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NL	Netherlands	VN	Viet Nam
CG	Congo	KE	Kenya	NO	Norway	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NZ	New Zealand	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	PL	Poland		
CM	Cameroon	KR	Republic of Korea	PT	Portugal		
CN	China	KZ	Kazakhstan	RO	Romania		
CU	Cuba	LC	Saint Lucia	RU	Russian Federation		
CZ	Czech Republic	LI	Liechtenstein	SD	Sudan		
DE	Germany	LK	Sri Lanka	SE	Sweden		
DK	Denmark	LR	Liberia	SG	Singapore		
EE	Estonia						

TITLE OF THE INVENTION
DISK CACHE ENHANCER WITH DYNAMICALLY SIZED
READ REQUEST BASED UPON CURRENT CACHE HIT RATE

BACKGROUND OF THE INVENTION

5 Cache is a storage area that keeps frequently accessed data or program instructions readily available so that data or program instructions (both referred to hereafter, as "data") used by a computer do not have to be repeatedly retrieved from a secondary storage area. In one typical scheme, cache is a form of random access memory (RAM) which can be directly and quickly accessed by the computer's processor. In
10 contrast to cache, a computer also includes one or more secondary storage areas, typically disk devices, which can only be accessed through an input/output (I/O) device, thereby providing much slower access time than the cache. Ideally, a computer would run entirely from data stored in RAM-type cache. However, RAM is very expensive relative to disk memory. Thus, a small amount of cache is provided in the computer (relative to
15 memory capacity of the disks) to improve overall performance of the computer.

 Fig. 1 shows a prior art computer system 10 which uses cache. The cache in Fig. 1 is "software cache" or "disk cache" which is designed to store data and program instructions which are frequently accessed from a disk drive or tape drive. The system 10 is shown with an application software program 12 running and issuing I/O requests to a
20 cache manager 14. The cache manager 14 retains a memory of data recently accessed from one or more physical disks, collectively referred to as disk 16, and which is stored in cache 18 within the cache manager 14. If the program 12 requests data which exists in

the cache 18, the data is retrieved directly from the cache 18 and no read request for the data is sent to the disk 16. However, if the requested data is not in the cache 18, the cache manager 14 issues an I/O read request to the disk 16, resulting in a seek and transfer of data from the disk 16 into the cache 18. Then, the cache manager 14 copies the data (now stored in the cache 18 for potential subsequent use) into the memory of the application program 12 (not shown) for immediate use.

In one conventional system 10, the cache 18 is divided into "pages" and the data in the cache 18 includes pages of one or more application program instructions and/or pages of data used by the one or more programs. When the application program 12 requests a page, and the page is not in the cache 18, a "page fault" occurs. Upon the occurrence of a "page fault," the cache manager 14 transmits a disk read request to the disk 16 to retrieve the page. The retrieved page is forwarded to the application program 12 and is cached for potential subsequent use.

The "cache hit rate" is a measure of the percentage of times that the requested data is available in the cache 18, and, thus, does not need to be retrieved from the disk 16. Disk drive life and program execution speed will improve as the cache hit rate increases, since read requests cause physical wear and since data access time from cache is typically significantly faster than data access time from a disk. Many schemes have been developed to optimize the disk cache process so as to minimize the number of seek and read requests for data stored on the disk 16. Some schemes affect how the cache 18 is "populated" or "primed" with data. Other schemes are used to decide which data should be purged from the cache 18 as the space in the cache 18 becomes filled. Still other schemes are used to decide how to share valuable computer RAM between virtual memory and disk cache. U.S. Patent No. 5,581,736 (Smith), which is incorporated by reference in its entirety herein, is one example of the latter scheme.

One conventional scheme to improve the cache hit rate is to pre-read additional, unrequested data whenever a disk read request occurs. More specifically, this scheme reads the requested data from the disk, as well as a small amount of additional data on the disk which follows the requested data. This scheme is based on the fact that

data which is physically stored on the disk after the requested data is oftentimes likely to be needed shortly after the requested data is needed. The amount of additional, unrequested data that is read from the disk is called the "read ahead size." One conventional disk caching subsystem provided in Microsoft Windows has a small, fixed read-ahead size which can be preset by the user up to a maximum value of 64 kilobytes (64 K). For example, if the read-ahead size is 64K and 10K of data must be retrieved from the disk because it is immediately needed and is not present in the cache, then 74K of data is retrieved and cached. The 74K of data consists of the requested 10K, plus the subsequent 64K of data on the disk. Likewise, if the read-ahead size is 64K and 100K of data must be retrieved from the disk because the 100K of data is immediately needed and is not present in the cache, then 164K of data is retrieved and cached. Some disadvantages of this scheme are as follows:

- (1) The maximum read-ahead size is very small, thereby limiting the amount of additional data that is pre-read into the cache for potential subsequent use.
- (2) The read-ahead size is fixed and thus cannot dynamically change based upon system performance.
- (3) The additional read data (i.e., read-ahead data) is always data which follows the requested data. In some instances, a program is likely to need data which precedes the requested data. In the conventional scheme, a separate disk read must be performed to obtain the preceding data unless the preceding data was coincidentally captured as part of the read-ahead data associated with a different prior disk read operation.

Despite the many schemes for improving and optimizing disk cache performance, there is still a need to further improve and optimize performance, and thus further reduce the number of disk read requests. The present invention fulfills this need.

BRIEF SUMMARY OF THE PRESENT INVENTION

A method is provided of reading data in a computer system, wherein the computer system includes a storage device and a cache in communication with the storage device. The method comprises tracking a cache hit rate of the computer system,

detecting a request for data which is immediately requested by the computer system but which is not currently present in the cache, formulating a read request to obtain the requested data from the storage device, and dynamically sizing the read request based upon the current cache hit rate. The size of the read request is related to the cache hit rate in a manner such that the size of the read request is reduced as the cache hit rate declines, and the size of the read request is increased as the cache hit rate increases. Short-term and long-term cache hit rates are tracked. The short-term cache hit rate is used to determine the reduction in the size of the read request, and the long-term cache hit rate is used to determine the increase in the size of the read request.

BRIEF DESCRIPTION OF THE DRAWINGS

The following detailed description of preferred embodiments of the invention would be better understood when read in conjunction with the appended drawings. For the purpose of illustrating the invention, there is shown in the drawings embodiments which are presently preferred. It should be understood, however that the invention is not limited to the precise arrangements and instrumentalities shown. In the drawings:

Fig. 1 is a schematic block diagram of a conventional disk caching scheme;

Fig. 2 is a schematic block diagram of a disk caching scheme in accordance with a first embodiment of the present invention;

Fig. 3 is a combined functional flowchart and schematic block diagram of the disk caching scheme of the present invention; and

Fig. 4 is a schematic block diagram of a disk caching scheme in accordance with a second embodiment of the present invention

DETAILED DESCRIPTION OF THE INVENTION

Certain terminology is used herein for convenience only and is not to be taken as a limitation on the present invention. In the drawings, the same reference numerals are employed for designating the same elements throughout the several figures.

Fig. 2 is a schematic block diagram of a disk caching scheme in accordance with a first embodiment of the present invention. Referring to Fig. 2, system 20 is similar in many respects to the conventional system 10 of Fig. 1, except that the system 20 includes an additional element, namely, a disk cache enhancer 22 (hereafter, DCE 22). The DCE 22 functions as an add-on device to the conventional system 10 and does not interfere with the normal operation of the conventional system 10. Instead, the DCE 22 generates additional commands in the form of second disk reads to the cache manager 14 to improve the hit rate of the cache 18. One conventional system 10 suitable for use with the system 20 is the disk cache subsystem of Windows 95, Windows 98, or Windows NT.

The DCE 22 includes a time stamp 24 for incoming read requests; a performance monitor 26 for tracking a long-term cache hit rate, a short-term cache hit rate, disk transfer rate, average seek time, and other statistics; and a second read requester 28 for initiating second disk read requests.

The system 20 operates as follows:

- (1) When the application program 12 needs data, a read request is transmitted from the program 12.
- (2) The DCE 22 receives the read request, time stamps the read request, and forwards the read request unchanged to the cache manager 14.
- (3) The cache manager 14 processes the read request in a conventional manner, as described above. Thus, if the requested data currently exists in the cache 18, the data is retrieved directly from the cache 18 and is sent to the memory of the application program 12 (not shown) for immediate use, and no read request for the data is sent to the disk 16. However, if the requested data is not in the cache 18, the cache manager 14 issues an I/O read request to the disk 16, resulting in a seek and transfer of data (referred to as "first data" in Fig. 2) from the disk 16 into the cache 18. Then, the cache manager 14 copies the data (now stored in the cache 18 for potential subsequent use) into the memory of the application program 12 (not shown) for immediate use. The read request transmitted by the cache manager 14 may also obtain additional data as part of the first data based upon

the preset read-ahead size, as discussed in the background section above.

(4) Shortly after execution of the operations in (3), the performance monitor 26 of the DCE 22 uses the read request time stamp and the arrival time of the data to determine if the current read request resulted in a cache hit or cache miss. A "hit" is detected if the data transmitted to the program 12 arrives faster than a preset time period (i.e., if the read is completed quickly), and a "miss" is detected if the data transmitted to the program 12 arrives slower than the preset time period (i.e., if the read is not completed quickly). The preset time period is based upon known cache access and disk access times. The "hit" or "miss" status, as well as the data arrival times are used to update the long-term cache hit rate, short-term cache hit rate, disk transfer rate, average seek time, and other statistics kept by the performance monitor 26.

(5) If a "hit" is detected in the DCE 22, no further action is taken by the DCE 22 other than to update the statistics.

(6) If a "miss" is detected by the DCE 22, the DCE 22 formulates a second logical disk read from the second read requester 28 and issues it to the cache manager 14 to prime the cache 18 and improve the subsequent hit rate. The second disk read is "dynamically sized" based upon the current cache hit rate. The "current cache hit rate" is a moving average of the cache hit rate over time. "Dynamically sized" means that the size of the second read request depends upon at least one variable factor other than the size of the requested data which is inherently variable. The second disk read is preferably a "read-around" request formulated to include the data requested in the original read request, plus additional data which is not immediately requested and which is located before the starting point and after the end point of the immediately requested data. The second read request thus differs in at least two significant ways from the conventional read request. First, the size of the additional data requested in the second read request is variable, instead of being fixed in the conventional scheme. Second, the data includes data located before and after the required data, instead of only after as in the conventional scheme. Furthermore, depending upon the current cache hit rate, the size of the additional data requested in the second read request will typically be significantly larger than the size of

additional data requested in a conventional scheme, and thus may alternatively be referred to as a "Big Read" (see Fig. 3). For example, the second read request might ask for an amount of data in the megabyte range, compared to a maximum of 64K in a conventional Windows scheme. In an alternative, but less preferred embodiment of the present invention, the second read request is a dynamically sized conventional read-ahead request, not a read-around request.

(7) The second disk read request is received by the cache manager 14 and processed in the same manner as a conventional read request. That is, the cache manager 14 checks to see if the data in the second disk read currently exists in the cache 18. If so, no read request for the data is sent to the disk 16 and no further action is taken by the cache manager 14 or the DCE 22. However, if all of the data in the second disk read is not currently in the cache 18, the cache manager 14 issues an I/O read request to the disk 16, resulting in a seek and transfer of data (labeled as "second data" in Fig. 2) from the disk 16 into the cache 18 for potential subsequent use. The cache manager 14 also forwards the data to the DCE 22 (the requester) as part of its normal protocol. The DCE 22 receives the data but does not store it.

In most disk drive implementations of the present invention, the second disk read request usually results in a cache miss and a subsequent disk read, since it is unlikely that the cache 18 contains all of the typically large amounts of additional requested data. The subsequent disk read typically occurs efficiently because the mechanical arm of the disk drive is already at or close to the desired reading location due to the previously executed disk read of data associated with the original read request.

As noted above, the dynamically sized second read request is based upon the current cache hit rate. More specifically, the size of the second read request is related to the cache hit rate in a manner such that the size of the read request is reduced as the cache hit rate declines, and the size of the read request is increased as the cache hit rate increases. Preferably, the size of the second read request is further dependent upon the short-term cache hit rate and the long-term cache hit rate wherein the short-term cache hit rate is used to determine the reduction in the size of the read request, and the long-term

cache hit rate is used to determine the increase in the size of the read request. The short-term/long-term scheme produces a hysteresis response in the second read request size and allows for more stable and rapid adaptation. One suitable algorithm for determining the size of the second read request, as expressed in C programming language derived directly from the source code Appendix below, is as follows:

```

int xn0; // current max readahead size from cache table
int xn; // current readahead size
int xnMax=Nmax; // current physical readahead buffer size
// if less than xn and Nmax, multiple [pre]reads are done qq[slower?]
10 int xlen=1<<Nmax, xmsk=-1<<Nmax; char *xbuf=0;
int fastaverage=0,slowaverage=0; // no misses
#define MissShft 30
#define one (1<<MissShft)

void xnset(int new_xn) {static int xn0_=0;
15 if (xn0_!=xnMax) {xn0_=xnMax; xfree(xbuf);
phys_buf_len= 1<<xnMax;
xbuf=malloc(phys_buf_len);
}
if (new_xn>xn0 || new_xn<1) return;
20 if (xn!=new_xn) {xn=new_xn; xlen=1<<xn; xmsk=-1<<xn;
if (debugging && postread) seeMiss();
}
}

int freePgs, unlockedPgs, cachePgs,chngs;
25 int bigread,bigreads;

int tab[]={14000,21, 13000,20, 12000,19, 11000,18, 9000,17, 0,0};

```

```

// first entry of each pair is cache size required for second entry readahead
// setting entry 1 to 0 always forces xn0 to second entry

//
void check_cache_size() { int i,p;
5   freePgs = GetFreePageCount(0,&unlockedPgs);
   cachePgs=VCache_GetSize(0,0);
   p=(cachePgs*4096)/K;
   for(i=0; tab[i]; i+=2) {if(p>=tab[i]) {xn0=tab[i+1]; break;}}
   if (xn0<Nmin) xn0=Nmin;
10  if (xn0<xn) xnset(xn0); // ok to be greater
}

#define MissSzMax 10
int max_miss[2][Nmax+1]; // max allowable miss rate for each readahead size
int min_miss[2][Nmax+1]; // increase readcheck_cache_size size if below this
15 int minMiss[MissSzMax],maxMiss[MissSzMax];
   // above are decayed averages of misses

int sensitivity=3;
void doMiss(int m) { int i,n=0; // m=0[hit?] or 1[miss]
   fastaverage=fastaverage-(fastaverage>>sensitivity);
20  slowaverage=slowaverage-(slowaverage>>(sensitivity+3));
   if (m) { // reduce readcheck_cache_size possibly
   fastaverage += one>>sensitivity;
   slowaverage += one>>(sensitivity+3);
   if (slowaverage>max_miss[0][xn]) {
25  if (xn>0) xnset(xn-1); // fight reducing
   }
}

```

```
    }
    else { // increase read ahead size possibly
        if (fastaverage < min_miss[0][xn]) xnset(xn+1); // quick increase
    }
5
}

void set_up_internal_tables() {int i;
    for ( i=0; i<=Nmax; i++) { // calculate miss rate thresholds
        max_miss[0][i]=one/4+one/(1+i);
10    min_miss[0][i]=max_miss[0][i]*2/3;
    }
    check_cache_size();
    xnset(xn0); //
}

15    int ramspeed=1; // 10*2^19 bytes per millisec
    void calibrate_cache_copy_time(length) {int n=1;
        char *buf1=0; int t1,t2;
        for (;length>=1024; length/=2, n*=2) {
            if (!buf1) buf1=(char*)malloc(length); // lots of ram
20            else {
                t1 = getTime();
                {int i=0; for( ; i<n; i++) {memcpy(buf1,buf1+length/2,length/2);} }
                t2 = getTime();
                ramspeed=(length*n)/(t2-t1);
25            }
        }
        xfree(buf1);
```

```

    }

    handle_the_io() {
        if (it_is_a_simple_read) {
            int sfn=pir->ir_sfn, pos=pir->ir_pos; // save stuff
            5   int iolenv, len, pos2=0, len2=0, err, pid=pir->ir_pid;
                ioreq io; pioreq p=io;
                io=pir; // copy the io request
                t0 = getTime(); // time stamp
                iolenv=pir->ir_length; // attempted
            10   ret = (*PrevHook)(pfn, fn, Drive, ResType, CodePage, pir); // do the io
                dt2 = getTime()-t0; // elapsed time
                len=pir->ir_length; err=pir->ir_error;
                dt0=1+2*len/ramspeed; // time it would take to copy from cache
                if ( dt2>dt0 && !err ) { // we have a miss
            15   pos2 = io.ir_pos&=xmsk; // adjust start for read_around
                if (xn>=Nmin) { // read more
                    io.ir_length = phys_buf_len; io.ir_data=xbuf;
                    (*PrevHook)(pfn, fn, Drive, ResType, CodePage, p); // do bigread
                }
            20   len2=io.ir_length; // what we [would have] reread in
                }
                check_cache_size();
                doMiss(len2!=0); // update fast and slow averages, adju
                }
            25   else { // just pass it on
                ret = (*PrevHook)(pfn, fn, Drive, ResType, CodePage, pir);
            }
        }
    }

```

When an application program 12 randomly reads small records in different areas of the disk 16, the hit rate declines and the pre-read size is reduced, eventually becoming zero. When the pre-read size is zero, the DCE 22 provides no performance improvement in the system 20. However, most application programs 20 read related data most of the time, and the present invention detects that situation and dynamically enlarges the pre-read amount to a large number (e.g., typically 1 megabyte) in comparison to conventional schemes, such as Windows built-in cache routine which allows for a fixed, user-selected pre-read size up to 64K.

The present invention is based on the theory that if large reads are resulting in a high cache hit rate, then the system should continue performing large reads and should even increase the size of the reads. If the even larger reads further increases the cache hit rate, then the system should try even larger reads, and so on. Likewise, if large reads are not resulting in a high cache hit rate, then the system should stop performing large reads, since the large reads consume system resources without providing any significant benefit. Simply stated, if the action has great results, do more of it, and if the action has poor results, do less of it.

Upon initiation of the system 20, the pre-read size is preferably set to about 1 megabyte. Since the DCE 22 does not affect the normal operation of the cache manager 14, the cache manager 14 continues to pre-read data according to the user preset value, even if the pre-read size output by the DCE 22 becomes reduced to zero as a result of a long period of a very low cache hit rate.

The read-around scheme preferably starts the read at a number of bytes which is the largest integral multiple of the read-ahead size that is less than or equal to the original I/O starting address. For example, if the original I/O starting address requested by the application program 12 is address 1,000,001, and the read-ahead size is currently 0.5 megabytes, then the addresses which are read for caching purposes are: 1,000,000 to 1,499,999. One advantage of this scheme is that the pre-read data always pieces together to create the desired file while minimizing overlapping portions.

Fig. 3 is a combined functional flowchart/schematic block diagram 30 of

the disk caching scheme of Fig. 2. In view of the discussion above, Fig. 3 is self-explanatory and thus is not described in further detail herein. However, it is noted that the short-term cache hit rate is referred to in Fig. 3 as the FAST moving average, and the long-term cache hit rate is referred to in Fig. 3 as the SLOW moving average.

5 PERFORMANCE RESULTS OF PRESENT INVENTION

An important industry performance measure is the Ziff-Davis "WinBench" test suite, which is widely quoted when comparing the cost performance of various manufacturers' personal computers. The present invention improves the performance of the "Business WinDisk" section of the WinBench tests from 30 to 100 percent, depending
10 on the available hardware resources. The most important resource is the amount of internal memory allocated for disk caching. More modern systems with faster clock speeds and faster disk transfer rates tend to show greater improvement in this regard due to the use of the present invention.

Another important improvement is in reduced program loading time for
15 large programs that are page faulted in. This is best explained in the context of Windows 95, Windows 98 and Windows NT. Consider, for example, the start-up of a 1 megabyte program. First, the operating system places information about the whole program in the page table, but the program itself is not read into memory. The operating system then attempts to execute the first instruction, causing a page fault. The missing page (which is
20 only 4K) is then read in from the disk cache. The program is allowed to run until another page fault occurs, and the process continues until an initial "working set" of pages is in virtual memory. Thereafter, the program can run with a relatively small number of page faults.

Significant time can be expended if the disk caching subsystem performs a
25 physical seek and a read for each page being faulted in. Windows therefore reads in a small fixed amount (under user control up to 64K) of additional data and places that data into the cache in case it is required shortly by another page fault. In this situation, the present invention detects the cache miss and reads in a larger section of the program,

including parts of the program preceding the page fault, forcing more data into the cache. This has the effect of reducing the loading time of programs such as Netscape by half in a typical configuration.

5 A further performance improvement is also caused by the present invention as a result of the improved hit rate. Windows also monitors the hit rate and adjusts the cache size, as described in U.S. Patent No. 5,581,736 (Smith). The present invention, when practiced with the Smith scheme, causes Windows to allocate more memory to the cache, thereby resulting in a further performance improvement.

10 HARD DISK RELIABILITY IMPROVEMENTS RESULTING FROM USE OF THE PRESENT INVENTION

The main failure mode of hard disks occurs during seeks. The present invention significantly reduces the number of seeks. For example, the Ziff-Davis benchmark previously mentioned contains snapshots of activity caused by common PC programs including MS-Office, Lotus, Excel, MS Word, PowerPoint and others. This
15 activity contains about 52,000 reads which normally causes 12,000 seeks. The present invention reduces the seeks to 2,140 seeks. The reduced seeks translates into a substantial improvement in disk lifetime.

The DCE 22 is preferably implemented as a software driver. The DCE 22 may be installed as a device driver with any Windows 95, Windows 98, Windows NT
20 operating system, or the like.

The present invention is particularly useful in computer applications that make extensive use of disk reads. However, the scope of the invention includes systems wherein the disk 16 is another form of a storage device, such as a tape drive. More generally, the storage device may be any type of memory which is associated with a
25 cache for the memory.

SECOND EMBODIMENT WITH DCE FUNCTIONS INTEGRATED INTO DISK CACHE SUBSYSTEM/CACHE MANAGER

In the first embodiment of the present invention described above and illustrated in Figs. 2 and 3, the DCE 22 operates independent of the cache manager 14 and thus is particularly suitable as an add-on or retrofit scheme. However, the functions of the DCE 22 would likely be performed more efficiently if they were integrated into the cache manager 14. Some advantages of an integrated scheme are as follows:

(1) The performance monitor statistics and the cache hit and miss detection functions of the conventional cache manager 14 can be directly used for determining the size of the variable read request, in place of the indirect scheme of Fig. 2 which is used to obtain the statistics and detect cache hits and misses. Thus, the time stamp 24 and performance monitor 26 of Fig. 2 may be eliminated.

(2) The first disk read can be dynamically sized based upon the algorithms described above. Thus, no second disk read or return of second data would be required and the second read requester 28 of Fig. 2 may be eliminated.

Fig. 4 shows an integrated system 32. The system 32 is generally similar to the system 10 of Fig. 1, except that cache performance monitor 34 tracks short-term and long-term cache hit rates wherein a conventional cache performance monitor tracks only one cache hit rate. Furthermore, the cache manager 14' includes a pre-read size calculator 36 to determine a dynamically sized disk read wherein a conventional cache manager 14 outputs a fixed size disk read.

Although the present invention is preferably used with software cache or disk cache, the present invention may also be used in conjunction with other types of cache, such as cache memory or memory cache. For example, hardware cache is cache memory on a disk drive controller or a disk drive. The hardware cache stores frequently accessed program instructions and data, as well as additional tracks of data that a program might need next. A computer can access required data much more quickly from the hardware cache than from the disk. The data in the hardware cache is delivered directly to an expansion bus. A memory cache, sometimes called a cache store or RAM cache, is a portion of memory made of high speed static RAM (SRAM) instead of the slower and cheaper dynamic RAM (DRAM). In memory caching, data and instructions are cached in

SRAM to minimize the need to access the slower DRAM. Memory caches may be internal (Level 1 (L1)) or external (Level 2 (L2)). In a memory cache scheme, the "storage device" would be the DRAM. The scheme disclosed in the present invention may be adopted for all of the above-noted caching processes.

5 The following Appendix is the source code for one suitable implementation of the first embodiment of the present invention.

 It will be recognized by those skilled in the art that changes may be made to the above-described embodiments of the invention without departing from the broad inventive concepts thereof. It is understood, therefore, that this invention is not limited to
10 the particular embodiments disclosed, but is intended to cover all modifications which are within the spirit and scope of the invention as defined by the appended claims.

 What is claimed is:

APPENDIX

```

/* This VxD can output information to the debug console
5/21 got 1200 with d211zc19m16. each miss caused xn to drop from 19
cache grew from about 10 to 14 during run
5/22 got 1350 with fixed multiple read code
got 1200 with c20m17 about 2.5 times the misses [2500]
reduced cache params gave 1300, 1450, 1550, 1640[log 800 misses]
old [shipped to compaq, asus] driver gives 900 1200 on first 2 runs
8/30 noticed we perform poorer for 'application development' tests
note that link libraries are large files that contain small binaries
9/1 2055 non cached opens found in ziff davis
zd got 612,775,1030,1290,1480,1550,1650,... cache from 10 to 19
*/
// -----
// Device preliminaries

#define DEVICE_MAIN

#include <vtoolsc.h>

Declare_Virtual_Device(LOS0)
#undef DEVICE_MAIN

// #include "apcx.h" // definitions common to app
// APCX.H - include file for Asynchronous Procedure Call example

// These definitions are used both by the calling app and the VxD

#define APCX_REGISTER CTL_CODE(FILE_DEVICE_UNKNOWN, 1,
METHOD_NEITHER, FILE_ANY_ACCESS)
#define APCX_RELEASEMEM CTL_CODE(FILE_DEVICE_UNKNOWN, 2,
METHOD_NEITHER, FILE_ANY_ACCESS)

// -----
// Static data

PVOID      OpenFileApc = 0;    // ring 3 address to call
THREADHANDLE TheThread = 0;    // thread in which ring 3 call runs
ppIFSFileHookFunc PrevHook;    // previous IFS hook
int found_BIOS; // if 1, we found the appropriate bios signature
int debugging=0; // if >0 emits out()'s to debugger and optional log file
int prered=1; // 0 means no prered

// -----
// Declare prototypes for control message handlers

```

```

DefineControlHandler(SYS_DYNAMIC_DEVICE_INIT, OnSysDynamicDeviceInit);
DefineControlHandler(SYS_DYNAMIC_DEVICE_EXIT, OnSysDynamicDeviceExit);
DefineControlHandler(W32_DEVICEIOCONTROL, OnW32Deviceiocontrol);
DefineControlHandler(DEVICE_INIT, OnDeviceInit);

// -----
// Routine to dispatch control messages to handlers
//
BOOL ControlDispatcher(
    DWORD dwControlMessage,
    DWORD EBX, DWORD EDX, DWORD ESI, DWORD EDI, DWORD ECX)
{
    START_CONTROL_DISPATCH
    ON_SYS_DYNAMIC_DEVICE_INIT(OnSysDynamicDeviceInit);
    ON_SYS_DYNAMIC_DEVICE_EXIT(OnSysDynamicDeviceExit);
    ON_W32_DEVICEIOCONTROL(OnW32Deviceiocontrol);
    ON_DEVICE_INIT(OnDeviceInit);
    END_CONTROL_DISPATCH
    return TRUE;
}

///#define XDEBUG 1

#ifdef XDEBUG // for dynamic testing with the apc
#define xtrap() trap()
#define breakpoint() __asm int 3
#define APC(s) {if (TheThread) {_VWIN32_QueueUserApc(OpenFileApc, (DWORD)cpyStr(s),
TheThread);}}
// qq this must be a dynamic dll to call the APC
#else
#define xtrap() {}
#define breakpoint() {if (debugging) __asm int 3}
#define APC(s) {} /* dont call if we're static */
#endif

void checkfilename(pioreq pir);

#define enterSync() Begin_Critical_Section(0)
#define exitSync() End_Critical_Section()

void xfree(void *x) {if (x!=0) {free(x);}}
void trap() {}
#define getTime() Get_System_Time()
int xTime() {int date,time;
    time=VTD_Get_Date_And_Time(&date);

```

```

return date*(24*60*60)+time/1000; // seconds since ?
}

char s[150];
int s_length=0;
int flushalways=0,flushnow=0,flushing=0,logging=0; // to file

#define logbufsize 49000
char *logbuf;
int log_buf_length=0;
int time,time0;
void out(char *str) {char s1[160];
    if (debugging) {int time=getTime();
        int sec=(time-time0)/1000;
        // int tenths=(time-time0-sec*1000)/100;
        int hundredths=(time-time0-sec*1000)/10;
        int milliseconds=time-time0-sec*1000;
        int s1_length=sprintf(s1,"%d.%03d %s\n",sec,milliseconds,str); //prepend time
        APC(str); // called iff XDEBUG
        dprintf("%s",s1); // pass to debugger [some are lost?qq]
        if (logging) {int pushlog,pushflush,pushbug;
            // enterSync(); //qq crashes on Alt tab. can these be nested?
            if (s1_length+log_buf_length>logbufsize-4 || flushing || flushnow) {
                static n=0;
                char file[16];
                sprintf(file,"c:\\gl#%d.log",n++);
                pushbug =debugging; debugging=0;
                pushflush=flushing; flushing=0;
                pushlog = logging; logging=0;
                if (log_buf_length)
                    xWrite(file,logbuf,log_buf_length);
                debugging=pushbug;
                flushing=pushflush; logging=pushlog;
                log_buf_length=0;
                flushnow=flushalways;
                if (flushing) {xfree(logbuf); logbuf=0;
                    logging=flushing=0;
                    return;}
            }
            log_buf_length += sprintf(logbuf+log_buf_length,"%s",s1);
            // exitSync();
        }
    }
}

```

```

char *cpyStr(char *s0) { char *s=(char*)malloc(strlen(s0)+1); strcpy(s,s0); return s;}

#define Freset -1
#define Fopen 0
#define Fread 1
#define Fwrite 2
#define Fclose 3

void fNam(int f, int n, char **p, int *we) {
    static int tabSz=0; static char **fTab=0; static int *wTab=0; enterSync();
    if(fTab==0) {fTab=(char**)malloc(0); wTab=(int*)malloc(0); }
    if(n>=tabSz) {int i,sz0=tabSz;
        while(n>=tabSz) { tabSz=(tabSz<<1)+1; }
        fTab=(char**)realloc(fTab,tabSz*sizeof(char*));
        wTab=(int*)realloc(wTab,tabSz*sizeof(int));
        for(i=sz0;i<tabSz;i++) {wTab[i]=0; fTab[i]=0;}
    }
    switch (f) {
        case Fopen: xfree(fTab[n]); fTab[n]=cpyStr(p[0]); we[0]=wTab[n]=0; break;
        case Fread: p[0]=fTab[n]; we[0]=wTab[n]; break;
        case Fwrite: p[0]=fTab[n]; we[0]=wTab[n]=1; break;
        case Fclose: xfree(fTab[n]); p[0]=fTab[n]=0; we[0]=wTab[n]=0; break;
        case Freset:
            if(fTab!=0) {int i;
                for(i=0;i<tabSz;i++) {
                    if(fTab[i]!=0) {dprintf("%d %s\n",i,(fTab[i]==0)?"":fTab[i]);}
                    xfree(fTab[i]); fTab[i]=0;
                }
                xfree(fTab); xfree(wTab);
            } break;
    }
    exitSync(); ////can't put return on this line MS bug!!!
}

int t0=-1; // last count emitted
#define IO 1
#ifdef IO
int xWrite(char *f,void *d,int n) {
    HANDLE h; int m=0; WORD err; BYTE act;
    h = R0_OpenCreateFile(FALSE,f,
        /*mode*/
        OPEN_ACCESS_WRITEONLY|OPEN_SHARE_DENYWRITE|OPEN_FLAGS_COMMIT,
        /*createAttr*/ ATTR_NORMAL,
        /*action*/ ACTION_IFEXISTS_TRUNCATE|ACTION_IFNOTEXISTS_CREATE,
        /*flags*/ 0,&err,&act

```

```

    );
    m = R0_WriteFile(FALSE,h,d,n,0,&err); // ,0 = offset
    R0_CloseFile(h,&err);
    return m;
}

/*
int hist[100];
void histInit() { int i; for(i=0;i<100;i++) {hist[i]=0;} }
void histInc(int i_) {
    int i=(i_>99)? 99: (i_<0)? 0: i_;
    hist[i]++;
}

void histOut() { int i,j; static char a[32]; Str h=newStr(""); ref(h);
    for(i=0;i<100;i++) {sprintf(a,"%02d, %010d\n",i,hist[i]); addStr(h,a);}
    xWrite("c:\\tmp\\hist.dat",s(h),lenStr(h));
}
*/
#else
void sout(char*s) { dprintf("%s",s); }
void xsout() {}
void histInit() {}
void histInc(int i_) {}
void histOut() {}
#endif

char* getPath(int d, pioreq pir){
    _QWORD x;
    //divide by 2 from "short" to "char"; 3: "C:" ... "/000"
    int sz=(pir->ir_ppath->pp_totalLength>>1)+3;
    char *p=(char *)malloc(sz);
    if (p!=NULL) {
        p[0]='A'-1+d; p[1]='.';
        UniToBCSPath(p+2,pir->ir_ppath->pp_elements,sz-3,BCS_OEM,&x);
    }
    return p;
}

#define Nmax 19 /* max allowable single readahead size [19=512k]*/
int Nmin=14; // min readahead [14=16384]
int xn0; // current max readahead size from cache table
int xn; // current readahead size
int xnMax=19; // current physical readahead buffer size
    // if less than xn and Nmax, multiple [pre]reads are done qq[slower?]

```

```

int xlen=1<<Nmax, xmsk=-1<<Nmax; char *xbuf=0;
int phys_buf_len; // read only this much into xbuf
int fastaverage=0, slowaverage=0; // no misses
#define MissShft 30
#define one (1<<MissShft)
void seeMiss();

void xnset(int new_xn) {static int xn0_=0;
  if (xn0_!=xnMax) {xn0_=xnMax; xfree(xbuf);
    phys_buf_len= 1<<xnMax;
    xbuf=malloc(phys_buf_len);
    // malloc above happens at init only, xfree then does nothing
  }
  if (new_xn>xn0 || new_xn<1) return;
  if (xn!=new_xn) {xn=new_xn; xlen=1<<xn; xmsk=-1<<xn;
    if (debugging) seeMiss();
  }
}

int freePgs, unlockedPgs, cachePgs;
#define pages() {freePgs = GctFreePageCount(0,&unlockedPgs);
  cachePgs=VCache_GetSize(0,0);}

//int tab[]={99000,19, 17000,19, 15000,18, 13000,17, 10000,16, 0,0};
int tab[]={99000,19, 13000,19, 11000,18, 9000,17, 7000,16, 0,0};
// first entry of each pair is cache size required for second entry readahead
// setting entry 1 to 0 always forces xn0 to second entry

void ahead() { int i,p;
  pages(); p=(cachePgs*4096)/1000;
  for(i=0;tab[i];i+=2) {if(p>tab[i]) {xn0=tab[i+1]; break;}}
  if (xn0<Nmin) xn0=Nmin;
  if (xn0<xn) xnset(xn0); // ok to be greater
}

#define MissSzMax 10
int missSz=7; // the number of different average miss rates we are tracking
int missTst=one/2;
// half = 50%missRatio; one/4 = 25%;
int max_miss[Nmax+1]; // allowable miss rate for each readahead size
int min_miss[Nmax+1]; // increase readahead size if below this
int aMiss[MissSzMax], minMiss[MissSzMax], maxMiss[MissSzMax];
// above are decayed averages of misses without concern for data length

int sensitivity=3;

```



```

int xlen=1<<Nmax, xmsk=-1<<Nmax; char *xbuf=0;
int phys_buf_len; // read only this much into xbuf
int fastaverage=0, slowaverage=0; // no misses
#define MissShft 30
#define one (1<<MissShft)
void seeMiss();

void xnset(int new_xn) {static int xn0_=0;
  if (xn0_!=xnMax) {xn0_=xnMax; xfree(xbuf);
    phys_buf_len= 1<<xnMax;
    xbuf=malloc(phys_buf_len);
    // malloc above happens at init only, xfree then does nothing
  }
  if (new_xn>xn0 || new_xn<1) return;
  if (xn!=new_xn) {xn=new_xn; xlen=1<<xn; xmsk=-1<<xn;
    if (debugging) seeMiss();
  }
}

int freePgs, unlockedPgs, cachePgs;
#define pages() {freePgs = GetFreePageCount(0,&unlockedPgs);
  cachePgs=VCache_GetSize(0,0);}

//int tab[]={99000,19, 17000,19, 15000,18, 13000,17, 10000,16, 0,0};
int tab[]={99000,19, 13000,19, 11000,18, 9000,17, 7000,16, 0,0};
// first entry of each pair is cache size required for second entry readahead
// setting entry 1 to 0 always forces xn0 to second entry

void ahead() { int i,p;
  pages(); p=(cachePgs*4096)/1000;
  for(i=0;tab[i];i+=2) {if(p>tab[i]) {xn0=tab[i+1]; break;}}
  if (xn0<Nmin) xn0=Nmin;
  if (xn0<xn) xnset(xn0); // ok to be greater
}

#define MissSzMax 10
int missSz=7; // the number of different average miss rates we are tracking
int missTst=one/2;
// half = 50%missRatio; one/4 = 25%;
int max_miss[Nmax+1]; // allowable miss rate for each readahead size
int min_miss[Nmax+1]; // increase readahead size if below this
int aMiss[MissSzMax], minMiss[MissSzMax], maxMiss[MissSzMax];
// above are decayed averages of misses without concern for data length

int sensitivity=3;

```

```

void doMiss(int m) { int i,n=0; // m=0[hit?] or 1[miss]
    fastaverage=fastaverage-(fastaverage>>sensitivity); //qq what constant?
    slowaverage=slowaverage-(slowaverage>>(sensitivity+3));
    if (m) { // reduce readahead?
        fastaverage += one>>sensitivity;
        slowaverage += one>>(sensitivity+3);
        if (slowaverage>max_miss[xn]) {
            // if (xn==Nmin) {} // qq turn off?
            // else xnset(xn-1); // fight reducing
            if (xn>0) xnset(xn-1); // fight reducing
        }
    }
    else { // increase readahead?
        if (fastaverage<min_miss[xn]) xnset(xn+1); // quick increase
    }

    /*** original method
    for (i=0;i<missSz;i++) {
        int n=aMiss[i];
        aMiss[i]=n-n-(n>>i)+(m<<(MissShft-i));
        if (n<minMiss[i]) {minMiss[i]=n;}
        if (n>maxMiss[i]) {maxMiss[i]=n;}
    }
    for (i=missSz-1;i>=0;i--) {if (aMiss[i]>missTst) {n=i; break;}}
    // find slowest decay with high miss rate [n=0 always fails if a miss]
    xnset(xn0-2*n); // change preread [qq 2?]
    ***/
}

int reset=1,opens,line,ctime,mtime,ptime,cnt,miss,cread,mread,pread,uncached;
#define S (one/100)
void seeMiss() {int i;
// for (i=0;i<missSz;i++)
{sprintf(s+10*i,"%4d,%4d", (minMiss[i]>>20)*1000/1024, (maxMiss[i]>>20)*1000/1024);}

    sprintf(s,"%d,%d .%02d[%d]>.%02d .%02d>.%02d",cnt,miss,max_miss[xn]/S,xn,
        slowaverage/S,fastaverage/S, min_miss[xn]/S);
    out(s);
}

void setMiss() {int i; for(i=0;i<missSz;i++) {minMiss[i]=1<<MissShft; maxMiss[i]=0; } }
void resetMiss() {int i; for(i=0;i<missSz;i++) {aMiss[i]=0; }; setMiss(); }

int memsz=0; void *membuf;
void memtst() {static int sz=0;

```

```

if (sz!=memsz) { sz=memsz; xfree(membuf);
membuf=malloc(memsz);
if(membuf==0){sz=memsz=0;}
}}

int eof=0;
int cache_always=0;
void clear() {int i; // can't call at init time
reset=opens=line=ctime=mtime=ptime=cnt=miss=cread=mread=pread=0;
uncached=cof=0;
time=time0=getTime(); t0=0;
for ( i=0; i<=Nmax; i++) { // calc miss rate thresholds
// these should depend on disk seek time versus transfer time
// we could test for each drive and have a table for each drive qq
max_miss[i]=one/4+one/(1+i);
min_miss[i]=max_miss[i]*2/3;
}
ahead(); // check cache size
xnset(xn0); resetMiss();
}

#define K 1000
#define M (K*K)
int dt=250; // delta count to emit data
int test_init=1;
int offset=0;
void show_data() {
if (test_init && debugging) {test_init=0;
if (found_BIOS) out("Authorized"); else out("Unauthorized");
// sprintf(s,"offset %x",offset); out(s);
out("eof xn0 xn ctime,mtime,ptime cnt,miss cread,mread,pread cache,free,unlocked");
}
// if (reset) clear();
if (t0+dt<=cnt) {t0=cnt;
s_length=sprintf(s,"%d %d,%d %d,%d,%d %d,%d %d,%d,%d %d,%d,%d"
,cof
,xn0,xn, ctime/K,mtime/K,ptime/K
,cnt,miss, cread/M,mread/M,pread/M
,(cachePgs*4096)/K/K,(freePgs*4096)/K/K,(unlockedPgs*4096)/K/K
);
++line;
out(s);
//seeMiss();
setMiss(); // resets min and max only
}
}

```

```

}

#define NDRV 32
int drv[NDRV]; //0,1,2 assumed 0
void driveTest() {
    WORD szSect=0,sectsClust=0,freeClust=0,totClust=0,err=0;
    int i;
    for(i=0;i<NDRV;i++) {drv[i]=0;}
    for(i=3;i<NDRV;i++) {
        R0_GetDiskFreeSpace((BYTE)i,&sectsClust,&freeClust,&szSect,&totClust,&err);
        if (err==0) {
            char *az="_ABCDEFGHIJKLMNOPQRSTUVWXYZabcdef",*yn="NY";
            int csz=szSect*sectsClust;
            if (freeClust>0 && szSect==512) { drv[i]=1; }
            sprintf(s,"drive=%c %c, total=%04dM, free=%04dM",
                az[i],yn[drv[i]],(totClust*csz)/1000000,(freeClust*csz)/1000000); out(s);
        }
    }
}

int ramspeed=1; // 10*2^19 bytes per millisec
void calibrate() {char s[20]; int n=10, xlen=1<<Nmax;
    char *buf1=(char*)malloc(xlen),*buf2; int t1,t2;
    if (!buf1) return;
    buf2=malloc(xlen);
    if (!buf2) {xfree(buf1); return;}
    t1 = getTime();
    {int i=0; for( ; i<n; i++) {memcpy(buf1,buf2,xlen);} }
    t2 = getTime(); free(buf1); free(buf2);
    if (t2==t1) ramspeed=99999; //qq log this event
    else ramspeed=(xlen*n)/(t2-t1);
    if (debugging) {sprintf(s,"RAM speed= %d",ramspeed); out(s);}
    if (ramspeed==0) ramspeed=1; //qq log this
    driveTest();
}

int test_bios() {char s[8]="AMibios";
    char s[8]="ASUSTeK";
    char *t=(char *)0xf0000;
    int i;
    for (i=0;i<0xffff9;i++) {if (*(t+i)==s[0] && *(t+i+1)==s[1]
        && *(t+i+2)==s[2]
        && *(t+i+3)==s[3]
        && *(t+i+5)==s[5]
        && *(t+i+6)==s[6]

```

```

    && *(t+i+7)==s[7]
    && *(t+i+4)==s[4]) {offset=i; return 1;}
};
return 0;}

```

```

/* -----
File System API Hook

```

This function is installed to hook all file system calls. Each time a file is opened, it allocates memory to store the name of the file being opened, forms the file name string, and queues an APC, passing the address of the file name string.

When the APC runs, it passes the address of the file name back to the VxD through DeviceIOControl in order to allow the VxD to deallocate the memory used to store the file name.

** Values for ir_flags for VFN_OPEN: from ifs.h

```

*/
// ACCESS_MODE_MASK      0x0007  /* Mask for access mode bits */
// ACCESS_READONLY        0x0000  /* open for read-only access */
// ACCESS_WRITEONLY       0x0001  /* open for write-only access */
// ACCESS_READWRITE       0x0002  /* open for read and write access */
// ACCESS_EXECUTE         0x0003  /* open for execute access */

// SHARE_MODE_MASK        0x0070  /* Mask for share mode bits */
// SHARE_COMPATIBILITY     0x0000  /* open in compatibility mode */
// SHARE_DENYREADWRITE     0x0010  /* open for exclusive access */
// SHARE_DENYWRITE         0x0020  /* open allowing read-only access */
// SHARE_DENYREAD          0x0030  /* open allowing write-only access */
// SHARE_DENYNONE          0x0040  /* open allowing other processes access */
// SHARE_FCB               0x0070  /* FCB mode open */

```

/** Values for ir_options for VFN_OPEN: *

```

// ACTION_MASK           0xff  /* Open Actions Mask */
// ACTION_OPENEXISTING    0x01  /* open an existing file */
// ACTION_REPLACEEXISTING 0x02  /* open existing file and set length */
// ACTION_CREATENEW       0x10  /* create a new file, fail if exists */
// ACTION_OPENALWAYS      0x11  /* open file, create if does not exist */
// ACTION_CREATEALWAYS    0x12  /* create a new file, even if it exists */

```

/** Alternate method: bit assignments for the above values: */

```

// ACTION_EXISTS_OPEN    0x01    // BIT: If file exists, open file
// ACTION_TRUNCATE       0x02    // BIT: Truncate file
// ACTION_NEXISTS_CREATE 0x10    // BIT: If file does not exist, create

/* these mode flags are passed in via ifs_options to VFN_OPEN */

// OPEN_FLAGS_NOINHERIT    0x0080
// OPEN_FLAGS_NO_CACHE     R0_NO_CACHE /* 0x0100 */
// OPEN_FLAGS_NO_COMPRESS  0x0200
// OPEN_FLAGS_ALIAS_HINT   0x0400
// OPEN_FLAGS_NOCRITERR     0x2000
// OPEN_FLAGS_COMMIT        0x4000
// OPEN_FLAGS_REOPEN        0x0800    /* file is being reopened on vol lock */

/** Values returned by VFN_OPEN for action taken: */
// ACTION_OPENED          1        /* existing file has been opened */
// ACTION_CREATED          2        /* new file has been created */
// ACTION_REPLACED         3        /* existing file has been replaced */

int _cdecl ifsHook(pIFSFunc pfn, int fn, int Drive, int ResType,
    int CodePage, pioreq pir) {
    int ret=0, sfn=pir->ir_sfn, opt=pir->ir_options; char *path=0;
    if (ResType==IFSFH_RES_LOCAL && fn==IFSFN_OPEN) {
        enterSync(); checkfilename(pir); exitSync();
    }
    if (ResType==IFSFH_RES_LOCAL && fn==IFSFN_READ && opt==0) { static int reset=1;
        if (reset) {reset=0; clear(); calibrate();}
    }
    ///READ & opt==0 avoid mallocs on paging operations
    //qq if Drive>32??
    if (drv[Drive] && ResType==IFSFH_RES_LOCAL && fn==IFSFN_READ && opt==0) {
        int t1,t2,t3,dt0,dt2,dt3,spd;
        int sfn=pir->ir_sfn, pos=pir->ir_pos;
        int len0,len,pos2=0,len2=0,err, pid=pir->ir_pid;
        static ioreq io; static pioreq p=&io;
        enterSync();
        p[0]=pir[0];
        t1 = getTime();
        len0=pir->ir_length;
        ret = (*PrevHook)(pfn, fn, Drive, ResType, CodePage, pir);
        t2 = getTime(); dt2=t2-t1;
        len=pir->ir_length; err=pir->ir_error;
        dt0=1+2*len/ramspeed; // ram to ram copy time
        //qq if length read in < length attempted (eof) , no pre-read
    }
}

```

```

if ( dt2>dt0 && err==0 && len!=12 ) {
    int at_eof=0;
    if (len!=len0) {++eof; ++at_eof;}
    // if (len!=len0 && pos2==pos) dont waste time rereading? [about 3% of zd's ios]
    pos2 = (p->ir_pos&=xmsk); p->ir_length=phys_buf_len; p->ir_data=xbuf;
    if (preread && xn>=Nmin) {
        (*PrevHook)(pfn, fn, Drive, ResType, CodePage, p);
        pread += p->ir_length;
        if (xlen>phys_buf_len) { //qq [slower!!]
            p->ir_pos += phys_buf_len; // next seg
            (*PrevHook)(pfn, fn, Drive, ResType, CodePage, p);
            pread += p->ir_length;
        }
    }
    len2=p->ir_length; // what we [would have] reread in qq
    miss++; mread+=len; mtime+=dt2;
}
t3 = getTime(); dt3=t3-t2;
ctime+=dt2; ptime+=dt3; cnt++; cread+=len; time=t3;
ahead(); doMiss(len2!=0);
if (debugging) show_data();
exitSync();
} else {
    ret = (*PrevHook)(pfn, fn, Drive, ResType, CodePage, pir);
}
#endif
//don't execute this
if (0 && ResType==IFSFH_RES_LOCAL && !(fn==IFSFN_WRITE && opt!=0)) {
    int we=0;
    enterSync();
    switch (fn) { // Branch on file system operation
        case IFSFN_OPEN:
            path=getPath(Drive,pir);
            if (pir->ir_error==0) {fNam(Fopen,sfn,&path,&we);}
            xfree(path);
            fNam(Fread,sfn,&path,&we); // means ?? qq
            break;
        case IFSFN_READ:
            fNam(Fread,sfn,&path,&we);
            break;
        case IFSFN_WRITE:
            fNam(Fwrite,sfn,&path,&we);
            break;
        case IFSFN_CLOSE:
            ///fNam(Fread,sfn,&path,&we); if (path!=0 && we!=0) {out(path);}
    }
}

```

```

        fNam(Fclose,sfn,&path,&we);
        default: break;
    }
    exitSync();
}
#endif
return ret;
}

// -----
// Handler for W32_DEVICEIOCONTROL
//
// This function is called when
//
// (1) The app calls CreateFile
// (2) The app calls DeviceIOControl
// (3) The app exits (or calls CloseHandle)

DWORD OnW32Deviceiocontrol(PIOCTLPARAMS p) { DWORD status;
// Structure member dioc_IOCTLCode determines function.
switch (p->dioc_IOCTLCode) {
    case APCX_REGISTER:
        // When the app registers, grab the APC function address from the input
        // input buffer. Store the current ring 0 thread handle.
        OpenFileApc = *(PVOID*)p->dioc_InBuf;
        TheThread = Get_Cur_Thread_Handle();
    case DIOC_OPEN:        // CreateFile
    case DIOC_CLOSEHANDLE: // file closed
        status = 0; break; // return OK
    case APCX_RELEASEMEM:
        // The APC function calls DeviceIOControl when it is done with the file name
        // that was passed to it. The VxD frees the memory that was earlier
        // allocated.
        free(*(PVOID*)p->dioc_InBuf);
        status = 0; break;
    default:// Fail any other calls.
        status = 0xffffffff;
    }
    return status;
}

int enable=0;
void init() {
    //12/31/96 = 6209; 6/1/97 = 6361
    enable = ((xTime()/86400)<6361+30+31+31+30+31+30+31)? 1: 0; // dec 31
}

```



```

found_BIOS = test_bios();
if (found_BIOS) enable=1;
//OpenFileApc = 0;
if (enable) {
    //breakPoint();
    PrevHook = IFSMgr_InstallFileSystemApiHook(ifsHook);
}
}

void *waste; // for testing reduced memory behavior
void exit() {
    fNam(Freset,0,0,0); xfree(membuf);
    xfree(waste);
    OpenFileApc = 0;
    if (enable) {
        xfree(xbuf);
        IFSMgr_RemoveFileSystemApiHook(ifsHook);
    }
}

BOOL OnSysDynamicDeviceInit() { init(); return TRUE; }
BOOL OnSysDynamicDeviceExit() { exit(); return TRUE; }
BOOL OnDeviceInit(VMHANDLE hVM, PCHAR CommandTail){ init(); return TRUE; }

int getPath2( pioreq pir, char *p, int max){
    _QWORD x;
    //divide by 2 from "short" to "char";
    int sz=(pir->ir_ppath->pp_totalLength>>1)+1; // include NUL at end
    if (sz>max) sz=max; // truncate strange name
    UniToBCSPath(p,pir->ir_ppath->pp_elements,sz,BCS_OEM,&x);
    return sz;
}

char *p;
int value;
int getpair() { // parse capital letter, optional signed number
    int ch=*p++,sign=0;
    value=0;
    if (ch=='-') {++sign; ch=*p++;}
    if (ch>='A' && ch<='Z')
        while (*p>='0' && *p<='9') {value*=10; value+=*p++-'0';};
    if (sign) value=-value;
    return ch;
}

```

```

void checkfilename(pioreq pir) { int ch;
    static char testname[3]="#G1"; // suffix is a control input
    char path[128]; int length;
    if (reset) clear();
    length=getPath2(pir,path,127);
    ++opens;
    p=path+1; // skip '\
    if (debugging>1) {
        int opt=pir->ir_options;
        if (*p != '~' || (opt>2 && opt!=0x112 && opt!=0x12
            && opt!=0x10) )
            {sprintf(s,"open#%d %s %x",opens,path,opt); out(s);}
    }
    if (pir->ir_options&OPEN_FLAGS_NO_CACHE) {++uncached;
        if (cache_always) pir->ir_options -= 0x100;
    }
    if (*p++==testname[0] && *p++==testname[1] &&
        *p++==testname[2]) {int done=0;
        // rest of filename is our input
        while (!done) switch (ch=getpair()) {
            case 'B': cache_always=value; break;
            case 'C':
                // c0 says use normal cache table
                // cn sets upper pre-read amount
                // c:>type \#glrlw0c5 to force 512k always
                // c:>type \#glw7c0 to restore
                tab[0]=99000; // assume
                if (value<=Nmax) { // ignore cache size
                    tab[0]=1; tab[1]=value;
                }
            case 'D': debugging=value; break; //emits dprintf [log]
            case 'L': //write to \g1#N.log
                if (!value && logging) {flushing=1;
                    break;}
                if (!logging) logbuf=malloc(logbufsize);
                if (!logbuf) value=0;
                if (value>1) dt=value;
                test_init=1;
                logging=value; break;
            case 'M': if (value<=Nmax) {
                Nmin=value;
                if (xn<Nmin) xnset(Nmin);}
                break;
            case 'Q': calibrate(); break;
            case 'R': pre-read=value; //r0 means no pre-read

```

```

        break;
    case 'S': sensitivity=value; break;
    case 'V': // view data now
        sprintf(s,"%d non cached",uncached); out(s);
        flushalways=value;
        flushnow=1;
        t0=-9999; show_data(); break;
    case 'W': missSz=value; break;
        // W0 eliminates adaptation, W7 is max
    case 'X': // allocate memory of value*megs
        xfree(waste);
        if (value) {waste=malloc(value<<20);
            if (!waste) out("not enough free mem");
        }
        else waste=0;
        break;
    case 'Z': clear(); break;
    default: done=1;
}
//brk: __asm int 3
    if (debugging) {sprintf(s,"%s",path); out(s);}
}

```

/*****
 the following is [supposed to?] work at the debug dot prompt
 Use Soft-ICE/W or WDEB386 to interact with this VxD.

Function

dgets - get a string from the debug console

Input

buf buffer to receive string
 maxchar number of bytes not including terminating nul that
 buf can accommodate

Returns

Returns the number of characters read
 console input is terminated by a CR. changed to NUL

/*****

```

int dgets(char* buf, int maxchar) { int i;
    for (i=0; i < maxchar; i++, buf++)
        { WORD ch;
            *buf = ch = In_Debug_Chr() & 0xff;

```

```

        if ( ((ch & 0xff) == 13) || ((ch & 0xff00) == 0xff00) )
            break;
// mishandles backspace, other ctl keys
        else Out_Debug_Chr((BYTE)(ch & 0xff));
    }
    *buf = 0;
    return i; // number of chars read
}

// Function
//   OnDebugQuery

// Remarks
//   Allows interactive control

//   Invoke this by typing .drivename at the debugger prompt

VOID OnDebugQuery() {
    CHAR buf[80]; INT index; BYTE statreg;

    dprintf("Enter on/off state [0=off, else on]: ");
    dgets(buf, sizeof(buf)-1);

    sscanf(buf, "%d", &index);
    //running = index;
}

```

/*.....\

to run: just add the line 'device=c:\directory\ios0.vxd' after [386Enh]
in \windows\system.ini

to alter the behavior:

- 1: go to a DOS prompt
- 2: issue a DOS command which tries to open a special, nonexistant
file name in the following form:

c:\> type \#glxxxx

The driver will see the attempted open of \#glxxxx and interpret the xxxx
as a command. Some examples:

To disable any prereading:

type \#glr0

To [re]enable prereading:

type \#glrl

To begin to emit debugging information:

type `\#gldl`

To use table of maximum allowable readahead depending on current cache size:

type `\#glc0`

[this reduces prereading and possible thrashing when cache size becomes too small. It is the default]

To change the maximum preread amount: [ignoring cache size]

type `\#glcN`

where N is from 1 to 21. N=21 sets max=2 meg, N=20 for max 1 meg, ... down to 16 normally [64k preread]

To change the minimum preread amount:

type `\#glmN`

where N is as above.

The driver detects when the prereading is not justified by a low average miss rate, for example when random reads are occurring. The readahead size is then reduced. Conversely, when good performance is observed, the readahead size is increased.

To change the miss rate sensitivity.

type `\#glSN`

Where N=0 to 7, 0 is fastest response; 4 is default

To set aside memory [for testing reduced memory configurations]

type `\#glxN`

where N=the number of megs of ram that will be reserved. N=0 is default. If unsuccessful, a fail message is logged.

To cause log data to be written to `c:\gl#nn.log`

type `\#GIDILN`

files written will have consecutive 'nn' numbers starting with 0.

They are normally written out after about 50k of log data have been accumulated.

Data will be included in the log file normally after every 250 reads.

If N is not 1, then data will be included every N reads.

To flush out whatever data has been accumulated, writing out a new log file:

type `\#glV`

The driver commands can all be combined, as in:

type `\#gld0c19r1m19x0`

to always read ahead 512k, no debugging, no memory wasted. The 0 is optional, so the following is equivalent.

type \#gldc19m19x

To stop the logging and write out the remaining data.
type \#G1L

To emit configuration info [if logging or debugging]
type \#g1q

To clear data and start from 0.
type \#g1z

When a situation is observed that merits investigation, try
type \#g1d21100vqz
to take a look at the current data every 100 reads, calibrate, reset counters

type \#g1vl
to include the final data, and close the log file.

[to mislead zd Benchmark by forcing all opens to be cached, (like
Intel's driver does).
include a 'bl' in the filename. 'b0' resets.]

*****/

CLAIMS

1. A method of reading data in a computer system, the computer system including a storage device and a cache in communication with the storage device, the method comprising:
 - (a) tracking a cache hit rate of the computer system;
 - (b) detecting a request for data which is immediately requested by the computer system but which is not currently present in the cache;
 - (c) formulating a read request to obtain the requested data from the storage device; and
 - (d) dynamically sizing the read request based upon the current cache hit rate.
2. A method according to claim 1 wherein in step (c), the size of the read request is at least initially greater than the size of the immediately requested data.
3. A method according to claim 2 wherein the initial size of the read request is about 1 megabyte greater than the size of the immediately requested data.
4. A method according to claim 2 wherein in step (c), the read request is a read-ahead request formulated to obtain the immediately requested data, plus additional data which is not immediately requested and which is located adjacent to the immediately requested data.
5. A method according to claim 2 wherein in step (c), the read request is a read-around request formulated to obtain the immediately requested data, plus additional data which is not immediately requested and which is located before and after the immediately requested data.
6. A method according to claim 1 wherein in step (d), the size of the read request is related to the cache hit rate in a manner such that the size of the read request is reduced as the cache hit rate declines, and the size of the read request is increased as the cache hit rate increases.

7. A method according to claim 6 wherein step (a) includes tracking a short-term and long-term cache hit rate, the size of the read request in step (d) being further dependent upon the short-term cache hit rate and the long-term cache hit rate, the short-term hit rate being used as the current cache hit rate to determine the reduction in the size of the read request, and the long-term cache hit rate being used as the current cache hit rate to determine the increase in the size of the read request.

8. A method according to claim 1 wherein the computer system runs an application program which makes first read requests whenever it needs data to execute the program, the method further comprising:

(e) receiving the first read request in a cache enhancer and in a cache manager, the cache manager including cache;

(f) providing the requested data to the application program from the cache if the requested data is currently in the cache; and

(g) formulating a second read request by the cache enhancer if the requested data is detected as not currently being in the cache, wherein the second read request is the dynamically sized read request of steps (c) and (d).

9. A method according to claim 8 wherein the detecting in step (g) is performed by monitoring the first read request response time.

10. A method according to claim 1 wherein the cache is divided into pages and the data includes pages of one or more program instructions and/or pages of data used by the one or more program instructions, and step (b) includes detecting a page fault.

11. A method according to claim 1 wherein the storage device is a disk and the cache is a disk cache.

12. A computer-readable medium whose contents cause a computer to read data in a computer system, the computer system including a storage device and a cache in communication with the storage device, by performing the steps of:

- (a) tracking a cache hit rate of the computer system;
- (b) detecting a request for data which is immediately requested by the computer system but which is not currently present in the cache;
- (c) formulating a read request to obtain the requested data from the storage device; and
- (d) dynamically sizing the read request based upon the current cache hit rate.

13. The computer-readable medium of claim 12 wherein in step (c), the size of the read request is at least initially greater than the size of the immediately requested data.

14. The computer-readable medium of claim 13 wherein the initial size of the read request is about 1 megabyte greater than the size of the immediately requested data.

15. The computer-readable medium of claim 13 wherein in step (c), the read request is a read-ahead request formulated to obtain the immediately requested data, plus additional data which is not immediately requested and which is located adjacent to the immediately requested data.

16. The computer-readable medium of claim 13 wherein in step (c), the read request is a read-around request formulated to obtain the immediately requested data, plus additional data which is not immediately requested and which is located before and after the immediately requested data.

17. The computer-readable medium of claim 12 wherein in step (d), the size of the read request is related to the cache hit rate in a manner such that the size of the read request is reduced as the cache hit rate declines, and the size of the read request is increased as the cache hit

rate increases.

18. The computer-readable medium of claim 17 wherein step (a) includes tracking a short-term and long-term cache hit rate, the size of the read request in step (d) being further dependent upon the short-term cache hit rate and the long-term cache hit rate, the short-term hit rate being used to determine the reduction in the size of the read request, and the long-term cache hit rate being used to determine the increase in the size of the read request.

19. The computer-readable medium of claim 12 wherein the computer system runs an application program which makes first read requests whenever it needs data to execute the program, the method further comprising:

(e) receiving the first read request in a cache enhancer and in a cache manager, the cache manager including cache;

(f) providing the requested data to the application program from the cache if the requested data is currently in the cache; and

(g) formulating a second read request by the cache enhancer if the requested data is detected as not currently being in the cache, wherein the second read request is the dynamically sized read request of steps (c) and (d).

20. The computer-readable medium of claim 19 wherein the detecting in step (g) is performed by monitoring the first read request response time.

21. The computer-readable medium of claim 12 wherein the cache is divided into pages and the data includes pages of one or more program instructions and/or pages of data used by the one or more program instructions, and step (b) includes detecting a page fault.

22. The computer-readable medium of claim 12 wherein the storage device is a disk and the cache is a disk cache.

23. An apparatus for reading data in a computer system, the computer system including a storage device and a cache in communication with the storage device, the apparatus comprising:

- (a) means for tracking a cache hit rate of the computer system;
- (b) means for detecting a request for data which is immediately requested by the computer system but which is not currently present in the cache;
- (c) means for formulating a read request to obtain the requested data from the storage device; and
- (d) means for dynamically sizing the read request based upon the current cache hit rate.

24. An apparatus according to claim 23 wherein the size of the read request is at least initially greater than the size of the immediately requested data.

25. An apparatus according to claim 24 wherein the initial size of the read request is about 1 megabyte greater than the size of the immediately requested data.

26. An apparatus according to claim 24 wherein the means for formulating a read request formulates a read-ahead request to obtain the immediately requested data, plus additional data which is not immediately requested and which is located adjacent to the immediately requested data.

27. An apparatus according to claim 24 wherein the means for formulating a read request formulates a read-around request to obtain the immediately requested data, plus additional data which is not immediately requested and which is located before and after the immediately requested data.

28. An apparatus according to claim 23 wherein the means for dynamically sizing the read request relates the read request to the cache hit rate in a manner such that the size of the

read request is reduced as the cache hit rate declines, and the size of the read request is increased as the cache hit rate increases.

29. An apparatus according to claim 28 wherein the means for tracking a cache hit rate includes means for tracking a short-term and long-term cache hit rate, the size of the read request being further dependent upon the short-term cache hit rate and the long-term cache hit rate, the short-term hit rate being used to determine the reduction in the size of the read request, and the long-term cache hit rate being used to determine the increase in the size of the read request.

30. An apparatus according to claim 23 wherein the computer system runs an application program which makes first read requests whenever it needs data to execute the program, the apparatus further comprising:

(e) means for receiving the first read request in a cache enhancer and in a cache manager, the cache manager including cache;

(f) means for providing the requested data to the application program from the cache if the requested data is currently in the cache; and

(g) means for formulating a second read request by the cache enhancer if the requested data is detected as not currently being in the cache, wherein the second read request is the dynamically sized read request.

31. An apparatus according to claim 30 wherein the means for detecting performs the detection by monitoring the first read request response time.

32. An apparatus according to claim 25 wherein the cache is divided into pages and the data includes pages of one or more program instructions and/or pages of data used by the one or more program instructions, and the means for detecting a request includes means for detecting a page fault.

33. An apparatus according to claim 25 wherein the storage device is a disk and the cache is a disk cache.

1/3

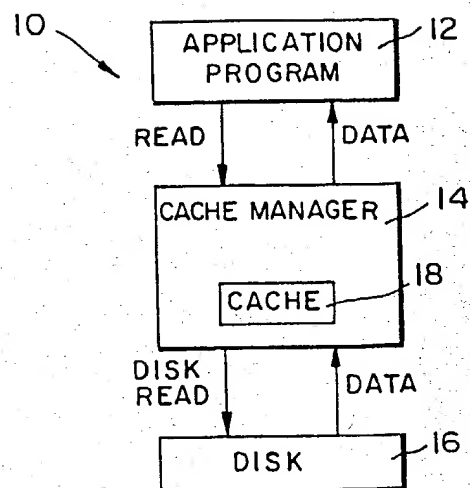


FIG. 1
(PRIOR ART)

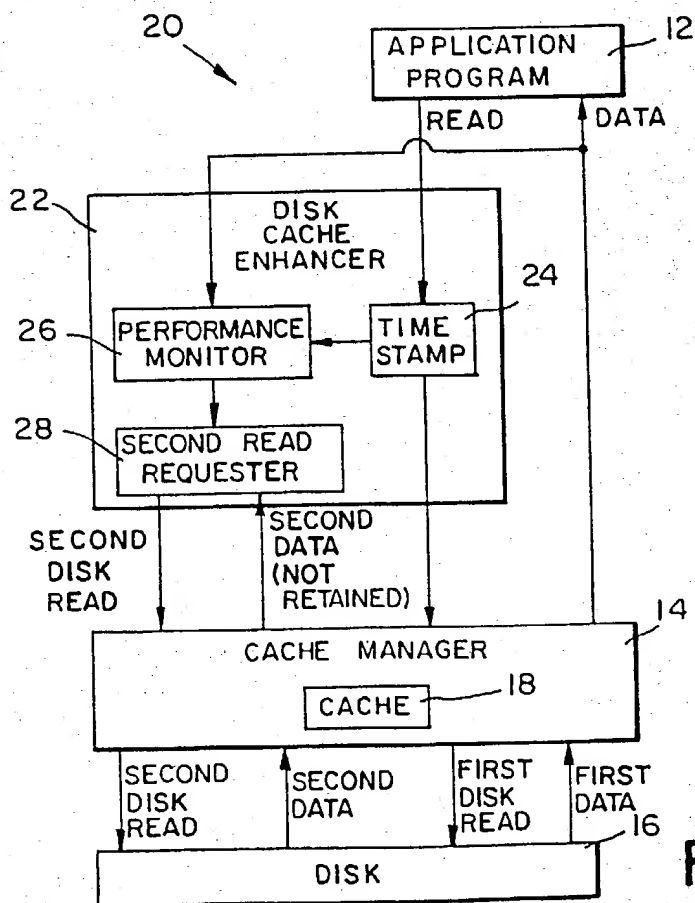


FIG. 2

2/3

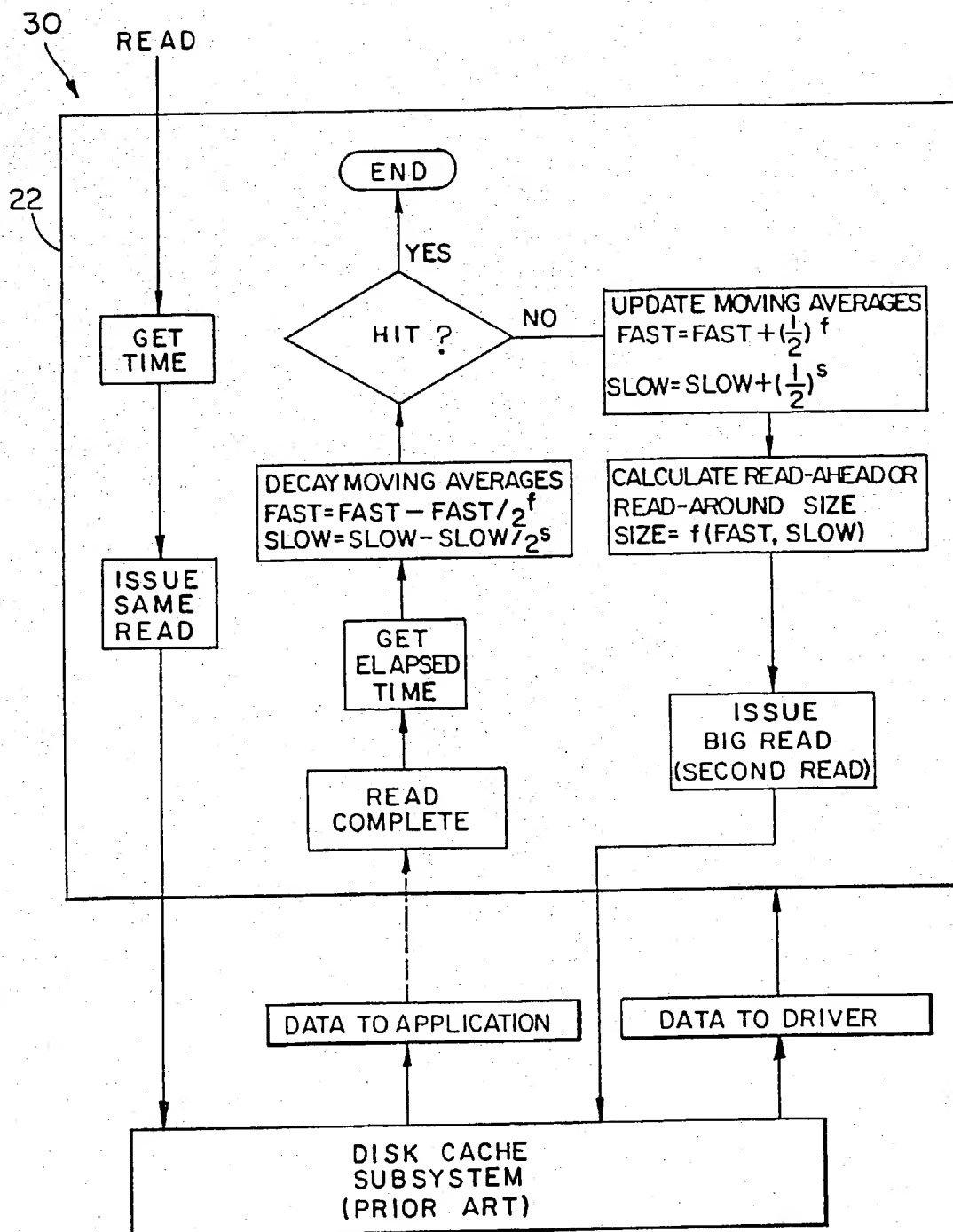


FIG. 3

SUBSTITUTE SHEET (RULE 26)

3/3

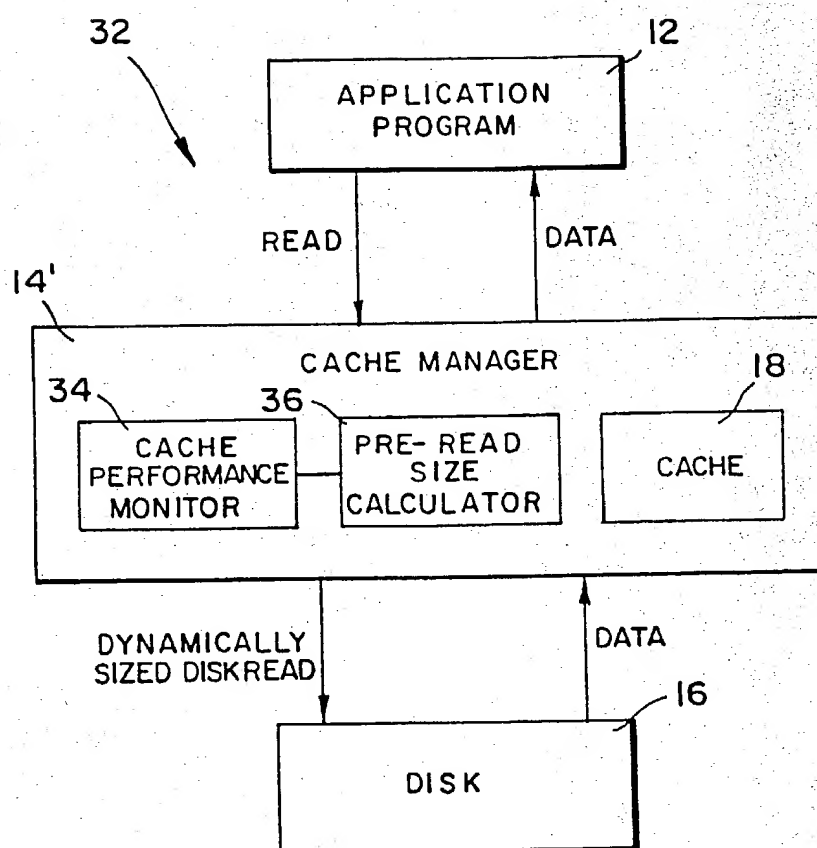
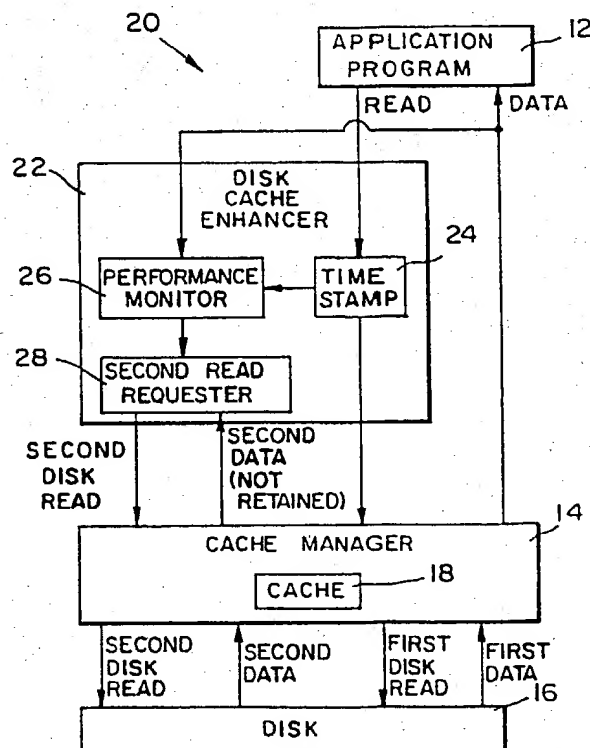


FIG. 4



INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification ⁶ : G06F 13/00	A3	(11) International Publication Number: WO 99/34356 (43) International Publication Date: 8 July 1999 (08.07.99)
(21) International Application Number: PCT/US98/27417 (22) International Filing Date: 23 December 1998 (23.12.98) (30) Priority Data: 60/070,195 30 December 1997 (30.12.97) US (71) Applicant: GENESIS ONE TECHNOLOGIES, INC. [US/US]; 7470 San Clemente Place, Boca Raton, FL 33433 (US). (72) Inventors: O'NEIL, John, T.; Apartment 3C, 613 Ocean Drive, Key Biscayne, FL 33149 (US). ISRAEL, Ben; 3040 N.W. 23rd Court, Boca Raton, FL 33431 (US). (74) Agents: JABLON, Clark, A. et al.; Panitch Schwarze Jacobs & Nadel, P.C., One Commerce Square, 22nd floor, 2005 Market Place, Philadelphia, PA 19103-7086 (US).		(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG). Published <i>With international search report.</i> <i>Before the expiration of the time limit for amending the claims</i> <i>and to be republished in the event of the receipt of amendments.</i> (88) Date of publication of the international search report: 7 October 1999 (07.10.99)
(54) Title: DISK CACHE ENHANCER WITH DYNAMICALLY SIZED READ REQUEST BASED UPON CURRENT CACHE HIT RATE		
(57) Abstract A computer system (20) including a disk cache subsystem having a disk cache enhancer (22) and an associated method for reading data from the computer system. The performance of a disk cache subsystem is enhanced by dynamically sizing read requests based upon the current cache hit rate or ratio. The size of the read request is reduced as the cache hit rate declines, and is increased as the cache hit rate increases. A short-term cache hit rate is tracked and used to determine the reduction in the size of the read request, and a long-term cache hit rate is used to determine the increase in the size of the read request. The read request may be a read-around request formulated to obtain the immediately requested data, plus additional prefetch data which is not requested and which is located before and after the immediately requested data.		



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakhstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US98/27417

A. CLASSIFICATION OF SUBJECT MATTER

IPC(6) : G06F 13/00

US CL : 711/113, 137

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)

U.S. : 711/112, 113, 114, 118, 137, 204, 213; 712/207, 237

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

USPTO Automated Patent System (APS), files USPAT, JPOABS, EPOABS

search terms: cache, prefetch, look ahead, read ahead, pre-read, hit, miss, rate, ratio, request, size

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 5,367,656 A (RYAN) 22 November 1994 (22.11.94), see column 2, lines 51-63; column 3, line 60 to column 4, line 4; and column 7, line 50 to column 9, line 7.	1-2, 4, 6, 8, 12-13, 15, 17, 19, 23, 24, 26, 28 and 30
X	US 5,649,153 A (MCNUTT ET AL) 15 July 1997 (15.07.97), see column 1, lines 54-61; column 2, lines 22-44; column 4, lines 50+; and column 7, lines 30-42.	1, 2, 4-6, 8, 11-13, 15-17, 19, 22-24, 26-28, 30 and 33
Y		3, 9-10, 14, 20-21, 25 and 31-32

☒ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

* Special categories of cited documents:	*T* later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
A document defining the general state of the art which is not considered to be of particular relevance	*X* document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
B earlier document published on or after the international filing date	*Y* document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
L document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	*&* document member of the same patent family
O document referring to an oral disclosure, use, exhibition or other means	
P document published prior to the international filing date but later than the priority date claimed	

Date of the actual completion of the international search

26 JULY 1999

Date of mailing of the international search report

17 AUG 1999

Name and mailing address of the ISA/US
Commissioner of Patents and Trademarks
Box PCT
Washington, D.C. 20231

Facsimile No. (703) 305-3230

Authorized officer

GLENN GOSSAGE

Telephone No. (703) 305-3900

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US98/27417

C (Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5,600,817 A (MACON, JR. ET AL) 04 February 1997 (04.02.97), see column 5, lines 47+.	1-33
A	US 5,146,578 A (ZANGENEHPOUR) 08 September 1992 (08.09.92), see column 3, line 4 to column 6, line 41 and column 7, lines 24-26.	1-33
A	US 5,325,509 A (LAUTZENHEISER) 28 June 1994 (28.06.94), see column 19, lines 52-55 and column 20, lines 10-20 and 39-50.	1-33
A	US 5,133,060 A (WEBER ET AL) 21 July 1992 (21.07.92), see entire document.	1-33
A	JP 2-18645 (SUDO) 22 January 1990 (22.01.90), see entire document.	1-33
A	JP 5-189286 A2 (AOKI ET AL) 30 July 1993 (30.07.93), see entire document.	1-33
A	JP 4-21043 (TERAYAMA) 24 January 1992 (24.01.92), see entire document.	1-33
A	US 5,581,736 A (SMITH) 03 December 1996 (03.12.96), see entire document.	1-33